Obliczenia Naukowe

Lista 1
Mateusz Kościelniak
nr indeksu: 244973

Zadanie 1

1.1 Epsilon maszynowy

- **Opis problemu**
  Napisanie programu w języku Julia wyznaczającego epsilon maszynowy (najmniejsza taka liczba $e>0$, że $1.0+e>1.0$) iteracyjnie dla wszystkich typów zmiennopozycyjnych i porównanie ich z wartościami zwracanymi przez funkcję $eps(type)$ z języka Julia oraz wartościami zawartymi w pliku nagłówkowym float.h języka C.

- **Rozwiązanie**

```
function machEps(type)
    macheps::type = 1
    while (type(1) + macheps / type(2)) > type(1)
        macheps /= type(2)
    end
    return macheps
end
```

- **Wyniki**

	algorytm	eps(type)	float.h
Float16	0.000977	0.000977	-
Float32	1.1920929e-7	1.1920929e-7	1.1920928955e-07
Float64	2.220446049250313e-16	2.220446049250313e-16	2.2204460493e-16

- **Wnioski**

  Widzimy, że dla każdego typu zmiennopozycyjnego, wyniki są takie same, co świadczy o tym, że algorytm podany przez mnie jest poprawny. Można zauważyć, że wartość epsilonu maszynowego maleje wraz ze wzrostem precyzji arytmetyki, a tą zależność można opisać wzorem 2^{-m} , gdzie m to długość mantysy w IEEE754, a całe wyrażenie to wartość epsilonu maszynowego.

1.2 Liczba eta

- **Opis problemu**

  Iteracyjne wyznaczenie liczby $eta>0$ i porównanie jej z funkcją $nextfloat(float\,number)$ z języka Julia, liczbą MIN_{sub} oraz przedstawienie zależności pomiędzy etą a epsilonem maszynowym.

- **Rozwiązanie**

```
function eta(type)
    eta::type = type(1)
        while eta / type(2) > type(0)
            eta /= type(2)
        end
        return eta
end
```

- **Wyniki**

	algorytm	nextfloat(type)	MIN_{sub}
Float16	6.0e-8	6.0e-8	-
Float32	1.0e-45	1.0e-45	1.4e-45
Float64	5.0e-324	5.0e-324	4.9e-324

- **Wnioski**

Tutaj również wyniki iteracyjnego obliczenia funkcji eta oraz wartości zwracane przez funkcję $nextfloat(type)$ są takie same. Wszystkie liczby mniejsze od liczby eta oraz większe od zera są traktowane jako zero. Po wykonaniu operacji $bitstring()$ w języku Julia na liczbie eta można zauważyć, że liczba ta jest równa najmniejszej liczbie subnormalnej, czyli liczbie mniejszej niż najmniejsza normalna liczba zmiennoprzecinkowa dodatnia.

1.3 Liczba max

- **Opis problemu**

Wyznaczenie liczby max – maksymalnej wartości dla każdego z typów zmiennopozycyjnych i porównanie wartości z funkcją $realmax(type)$ z języka Julia oraz wartościami przechowanymi w pliku nagłówkowym float.h języka C

- **Rozwiązanie**

```
function maxFlt(type)
    max::type = 2
    while !isinf(max * type(2))
            max *= type(2)
        end
    addend = max/type(2)

    while addend > eps(type)
                if !isinf(max + addend)
                    max += addend
                end
                addend /= type(2)
        end
        return max
end
```

- **Wyniki**

	algorytm	nextfloat(type)	float.h
Float16	6.55e4	6.55e4	-
Float32	3.4028235e38	3.4028235e38	3.4028235e38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e308

- **Wnioski**

Tak samo jak w poprzednich przykładach udało wyniki metody iteracyjnej pokrywają się z poprawnymi wartościami.

Zadanie 2

- **Opis**

Sprawdzenie czy wartość epsilonu maszynowego można aproksymować wyrażeniem Kahana $3*(4/3-1)-1$.

- **Rozwiązanie**

Implementacja wyrażenia Kahana dla różnych typów.

- **Wyniki**

	$3*(4/3-1)-1$	eps(type)
Float16	0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	2.220446049250313e-16	2.220446049250313e-16

- **Wnioski**

Wartości bezwzględne wyników uzyskane przy pomocy wyrażenia Kahana zgadzają się z poprawnymi wartościami epsilonu maszynowego, dla typów Float16 i Float64 różnią się jednak co do znaku, więc w wyliczeniach użyłem funkcji $\text{abs}()$. Wyrażenie to daje poprawne wyniki dzięki pewnej niedoskonałości komputerów które nie potrafią operować na liczbach rzeczywistych mających dowolną liczbę cyfr. Dokładność z jaką można te liczby przedstawiać, zależy od długości słowa w komputerze.

Zadanie 3

- **Opis**

  Sprawdzenie w języku Julia czy liczby są równo rozmieszczone w przedziale (1,2) z odległością $delta=2^{-52}$ oraz ich rozmieszczenie w przedziałach (½,1) i (2,4)

- **Rozwiązanie**

  Funkcja wyświetla reprezentacje bitową kolejnych n liczb oddalonych od siebie o delte poczynając od wartości min. Za pomocą tej funkcji i $deltą=2^{-52}$ badałem rozmieszczenie kolejnych liczb w wyznaczonych predziałach.

```
function deltaPrint(min, delta, n)
    for i in 1:n
        min += delta
        println(bitstring(min))
    end
end
```

- **Wyniki**

  Wyświetliłem po 5 liczb na początku oraz na końcu każdego przedziału. Widać że dla przedziału (1,2) delta wynosi 2^{-52}, po kilku eksperymentach z wielkością delty udało mi się ustalić, że dla przedziału (½,1) wynosi ona 2^{-53}, a dla przedziału (2,4) wynosi 2^{-51}.

```
od 1
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101
od 0.5
0011111111100000000000000000000000000000000000000000000000000010
0011111111100000000000000000000000000000000000000000000000000100
0011111111100000000000000000000000000000000000000000000000000110
0011111111100000000000000000000000000000000000000000000000001000
0011111111100000000000000000000000000000000000000000000000001010
od 2
0100000000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000
```

- **Wnioski**

  Liczby między kolejnymi potęgami dwójki są równomiernie rozmieszczone, a odległość pomiędzy kolejnymi liczbami w przedziałach (delta) podczas oddalania się od zera o kolejna potęgę maleje dwukrotnie. Dla wszystkich liczb w przedziale cecha jest identyczna, a zmianie ulega tylko mantysa, co za tym idzie ilość liczb w każdym przedziale można opisać wzorem 2^{m}, gdzie m to dlugość mantysy.

Zadanie 4

- **Opis**

  Napisanie programu znajdującego najmniejszą taką liczbę x w przedziale (1,2), że $x*(1/x)\neq 1$

- **Rozwiązanie**

  Algorytm zaczynał działanie od 1 i biorąc kolejne liczby w reprezentacji Floa64 sprawdzał czy posiadają pożądane własności, jeśli znajdował taką liczbę to przerywał działanie.

```
x = nextfloat(Float64(1))
while Float64(x * (Float64(1) / x)) != Float64(1)
    x = nextfloat(x)
end
```

- **Wyniki**

  Najmniejsza taka liczba to 1.0000000000000002.

- **Wnioski**

  Zaokrąglenie w arytmetyce zmiennopozycyjnej spowodowane skończoną ilością bitów używanych do reprezentacji liczby spowodowało, że znaleźliśmy liczbę która spełniała równanie bez rozwiązania. Przy używaniu typów zmiennopozycyjnych takie błędy są nieuniknione, ale możemy z nimi walczyć np. poprzez uproszczenie równania, które chcemy obliczyć w komputerze.

-

Zadanie 5

- **Opis**

  Obliczenie iloczynu skalarnego dwóch wektorów z wykorzystaniem 4 różnych algorytmów.

- **Rozwiązanie**

  Implementacja algorytmów:

  (a) “w przód” $\sum_{i=1}^{n} x_i y_i$
  (b) “w tył” $\sum_{i=n}^{1} x_i y_i$
  (c) od największego do najmniejszego (dodaj dodatnie liczby w porządku od największego do najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego, a następnie daj do siebie obliczone sumy częściowe),
  (d) od najmniejszego do największego (przeciwnie do metody (c))

- **Wyniki**

Wyniki:

	1	2	3	4
Float32	-0.4999443	-0.4543457	-0.5	-0.5
Float64	1.0251881368296672e-10	-1.5643308870494366e-10	0.0	0.0

Błędy:

	1	2	3	4
Float32	0.49994429944939167	0.4543457031149343	0.4999999999899343	0.4999999999899343
Float64	1.1258452438296672e-10	1.4636737800494365e-10	1.0065710700000004e-11	1.0065710700000004e-11

- **Wnioski**

To zadanie pokazuje, że kolejność działań ma znaczenie. Dodanie bardzo dużej liczby do bardzo małej generuje błędy. Jednym ze sposobów zmniejszenia błędów jest u życie arytmetyki o większej precyzji, choć jak widzimy w tym przypadku dalej nie dało to zadowalających rezultatów.

Zadanie 6

- **Opis**

Policzenie w arytmetyce Float64 wartości dwóch funkcji $f(x)=\sqrt{(x^2+1)}-1$ i $g(x)=x^2/(\sqrt{(x^2+1)}+1)$, gdzie f = g dla argumentu $x=b^{-i}$ $i \in \{1,2 \ldots n\}$.

- **Rozwiązanie**

Obliczanie wartości funkcji f oraz g dla kolejnych wartości w pętli po czym wypisanie wyników na ekran.

- **Wyniki**

8^x	f	g
-1	-0.030532544378698234	0.007933616752794135
⋮	⋮	⋮
-8	0.0	1.7763568394002568e-15
-9	0.0	2.7755575615628914e-17
⋮	⋮	⋮
-178	0.0	1.6e-322
-179	0.0	0.0

- **Wnioski**

Funkcje f oraz g dla malejącego argumentu dążą do zera, którego teoretycznie nigdy nie powinny osiągnąć, w praktyce robią to przez to, że komputer posiada ograniczoną arytmetykę. Funkcja f szybko uzyskuje zero, ponieważ odejmowane są bardzo bliskie obie liczby, funkcja g jest dużo dokładniejsza a jej błąd wynika w zasadzie z niedokładności arytmetyki.

Zadanie 7

- **Opis**

Obliczenie wartości pochodnej funkcji $f(x)=\sin(x)+\cos(3*x)$ w punkcie $x_0=1$ z definicji (dla h = {1,2,…,54} i normalnie, oraz obliczenie wartości błędu.

- **Rozwiązanie**

Przybliżoną wartość funkcji obliczałem wg wzoru $f'_{\sim}(x)=(f(x_0+h)-f(x_0))/h$, rzeczywistą wartość wg wzoru $f'(x)=\cos(x)-3*\sin(3*x)$, a błąd $|f'(x)-f'_{\sim}(x)|$

- **Wyniki**

$h=2^{-i}$	$f'_{\sim}(x)$	$\lvert f'(x)-f'_{\sim}(x)\rvert$	1+h
0	2.0179892252685967	1.9010469435800585	2.0
1	1.8704413979316472	1.753499116243109	1.5
$\vdots$	$\vdots$	$\vdots$	$\vdots$
52	-0.5	0.6169422816885382	1.0000000000000002
53	0.0	0.11694228168853815	1.0
54	0.0	0.11694228168853815	1.0

- **Wnioski**

Wartość 1+h=1 dla bardzo małych wartości h , to pokazuje, że należy unikać dodawania do siebie liczb które tak bardzo różnią się wykładnikami. Drugą rzeczą która zapewne wpływa negatywnie na wynik obliczeń jest odejmowanie $f(x_0+h)-f(x_0)$ czyli bardzo bliskich sobie liczb, co sprowadza się do utraty cyfr znaczących.